2-Stack Sorting is polynomial

Adeline Pierrot, Dominique Rossin

2-Stack Sorting is polynomial *

Adeline Pierrot Dominique Rossin

April 10, 2013

In this article, we give a polynomial algorithm to decide whether a given permutation σ is sortable with two stacks in series. This is indeed a longstanding open problem which was first introduced by Knuth in [1]. He introduced the stack sorting problem as well as permutation patterns which arises naturally when characterizing permutations that can be sorted with one stack. When several stacks in series are considered, few results are known. There are two main different problems. The first one is the complexity of deciding if a permutation is sortable or not, the second one being the characterization and the enumeration of those sortable permutations. We hereby prove that the first problem lies in P by giving a polynomial algorithm to solve it. This article strongly relies on [3] in which 2-stack pushall sorting is defined and studied.

1 Notations and definitions

Let I be a set of integers. A permutation of I is a bijection from I onto I. We write a permutation σ of I as the word $\sigma = \sigma_1\sigma_2\ldots\sigma_n$ where $\sigma_i = \sigma(i_1)$ with $I = \{i_1 \ldots i_n\}$ and $i_1 < i_2 < \cdots < i_n$. The size of the permutation is the integer n and if not precised, $I = [1\ldots n]$. Notice that given the word $\sigma_1\sigma_2\ldots\sigma_n$ we can deduce the set I and the map σ. For any subset J of I, $\sigma_{|J}$ denotes the permutation obtained by restricting σ to J. In particular the word corresponding to $\sigma_{|J}$ is a subword of the word corresponding to σ.

Let's recall the problem of sorting with two stacks in series. Given two stacks H and V in series –as shown in Figure 1– and a permutation σ, we want to sort elements of σ using the stacks. We write σ as the word $\sigma = \sigma_1\sigma_2\ldots\sigma_n$ with $\sigma_i = \sigma(i)$ and take it as input. Then we have three different operations:

- ρ which consist in pushing the next element of σ on the top of H.
- λ which transfer the topmost element of H on the top of V.
- μ which pop the topmost element of V and write it in the output.

*This work was completed with the support of the ANR project ANR BLAN-0204_07 MAGNUM

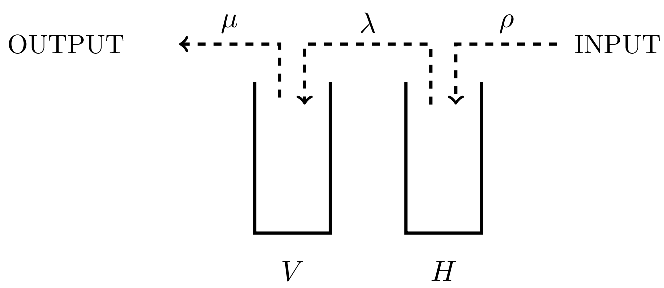
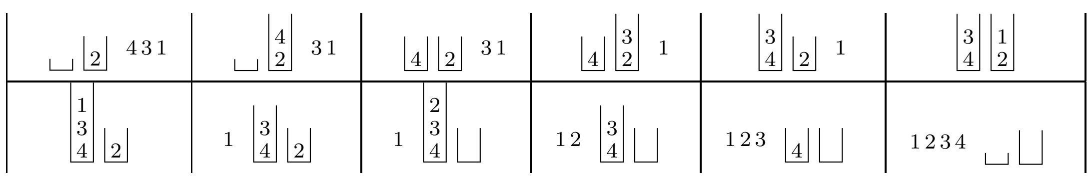


Figure 1: Sorting with two stacks in serie

If there exists a sequence $w = w_1 \dots w_k$ of operations ρ, λ, μ that leads to the identity in the output, we say that the permutation σ is 2-stack sortable. In that case, we define the sorting word associated to this sorting process as the word w on the alphabet $\{\rho, \lambda, \mu\}$. Notice that necessarily w has n times each letter ρ, λ and μ and $k = 3n$. For example, permutation 2431 is sortable using the following process.



This sorting is encoded by the word $w = \rho\rho\lambda\rho\lambda\rho\lambda\mu\lambda\mu\mu\mu$. We can also decorate the word to specify the element on which the operation is performed. The *decorated word* for w and 2431 is $\hat{w} = \rho_2\rho_4\lambda_4\rho_3\lambda_3\rho_1\lambda_1\mu_1\lambda_2\mu_2\mu_3\mu_4$. Note that we have the same information between (σ, w) and $\hat{w}$. Nevertheless, in a decorated word appears only once each letter ρ_i, λ_i or μ_i. The decorated word associated to (σ, w) is denoted $\hat{w}^\sigma$.

Of course not all permutations are sortable. The smallest non-sortable ones are of size 7, for instance $\sigma = 2435761$.

When only one stack is considered, there exists a natural algorithm to decide whether a permutation is sortable or not. Indeed, there is a unique way to sort a permutation using only one stack, and a greedy algorithm gives a decision procedure. For two stacks in series, a permutation can be sorted in numerous ways. Take for example permutation 4321. Each element can be pushed in either stacks H or V and output the identity at the end. Thus the decreasing permutation of size n has more than 2^n ways to be sorted *i.e.* more than 2^n sorting words.

Several articles introduce restrictions either on the rules or on the stack structure. For example, in his PhD-thesis West introduced a greedy model with decreasing stacks [4]. Permutations sortable with this model, called West-2-stack sortable permutations, are characterized and enumerated.

For our unrestricted case called sometimes in litterature *general 2-stack sorting problem*, no characterization of sortable permutations and no polynomial algorithm to decide if a permutation is sortable is known. A common mistake when trying to sort a given permutation is to pop out the smallest element i as soon as it lies in the stacks. This operation may indeed move other elements if i is not the topmost element of H. The elements above it are then transferred into V before i can be popped out. But sometimes, it can be necessary to take some elements of σ and push them onto H or V before this transfer. Take for example permutation 324617985. Trying to pop out the smallest element as soon as it is in the stacks leads to a dead-end. However, this permutation can be sorted using word $\rho_3\rho_2\lambda_2\rho_4\rho_6\rho_1\lambda_1\mu_1\mu_2\rho_7\lambda_7\lambda_6\lambda_4\lambda_3\mu_3\mu_4\rho_9\rho_8\rho_5\lambda_5\mu_5\mu_6\mu_7\lambda_8\mu_8\lambda_9\mu_9$. But we prove that this natural idea of popping out smallest elements as soon as possible can be adapted considering right-to-left minima of the permutation.

We saw that a sorting process can be described as a word on the alphabet $\{\rho, \lambda, \mu\}$. In this article, we will also describe a sorting in a different way. Take the prefix of a sorting word, it corresponds to move some elements from the permutation to the stacks or output them. At the end of the prefix some elements may be in the stacks. We can take a picture of the stacks and indeed, we will show that considering such pictures for all the prefixes that correspond to the entry of a right-to-left (RTL) minima of the permutation in H is sufficient to decide the sortability. Such a picture is called a stack configuration.

Definition 1. *A* stack configuration *c is a pair of vectors (v, w) of distinct integers such that the elements of v (resp. of w) corresponds to the elements of V (resp. of H) from bottom to top.*

A stack configuration is poppable *if elements in stacks H and V can be output in increasing order using operations λ and μ.*

Conditions for a stack configuration to be poppable have already been studied previously in [2, 3] and can be characterized by the following Lemma. Recall first that a permutation $\pi = \pi_1\pi_2 \dots \pi_k$ is a pattern of $\sigma = \sigma_1\sigma_2 \dots \sigma_n$ if and only if there exist indices $1 \leq i_1 < i_2 < \dots < i_k$ such that $\sigma_{i_1}\sigma_{i_2}\sigma_{i_3} \dots \sigma_{i_k}$ is order isomorphic to π.

Lemma 2. *A stack configuration c is poppable if and only if :*

- *Stack H does not contain pattern* 132*.*
- *Stack V does not contain pattern* 12*.*
- *Stacks (V, H) does not contain pattern* $|2|13|$*.*

Moreover, there is a unique way to pop the elements out in increasing order in terms of stack operations.

The first two conditions are usual pattern relation, considering elements in the stack from bottom to top. The third one means that there do not exist an element i in V and two elements j, k in H (k above j) such that $j < i < k$. There is a unique way to output those elements in increasing order as noticed in [3], so we will denote by $out_c(I)$ the word that consists in the operations necessary to output in increasing order elements of the set of values I from a stack configuration c.

Notice that a stack configuration has no restriction upon its elements except that they must be different. Most of the time, a stack configuration will be associated to a permutation implying that the elements in the stacks are a subset of those of the permutation. In particular a *total stack configuration of σ* is a stack configuration in which the elements of the stacks are exactly those of σ.

In this article we often use decomposition of permutations into blocks. A block B of a permutation $\sigma = \sigma_1\sigma_2 \dots \sigma_n$ is a factor $\sigma_i\sigma_{i+1} \dots \sigma_j$ of σ such that the set of values $\{\sigma_i, \dots, \sigma_j\}$ forms an interval. Notice that by definition of a factor, the set of indices $\{i, \dots, j\}$ also forms an interval. Given two blocks B and B' of σ, we say that $B < B'$ if and only if $\sigma_i < \sigma_j$ for all $\sigma_i \in B$, $\sigma_j \in B'$. A permutation σ is $\ominus$-decomposable if we can write it as $\sigma = B_1 \dots B_k$ such that $k \geq 2$ and for all i, $B_i > B_{i+1}$ in terms of blocks. Otherwise we say that σ is $\ominus$-indecomposable. When each B_i is $\ominus$-indecomposable, we write $\sigma = \ominus[B_1, \dots, B_k]$ and call it the $\ominus$-decomposition of σ. Notice that we do not renormalize elements of B_i thus except B_k, the B_i are not permutations. Nevertheless, B_i can be seen as a permutation by decreasing all its elements by $|B_{i+1}| + \dots + |B_k|$.

The RTL (right-to-left) minima of a permutations are elements σ_k such that there do not exist j respecting $j > k$ and $\sigma_j < \sigma_k$. We denote by σ_{k_i} the i^{th} right-to-left (RTL) minima of σ. If σ has r RTL minima, then $\sigma = \dots \sigma_{k_1} \dots \sigma_{k_2} \dots \sigma_{k_r}$ with $\sigma_{k_1} = 1$ and $k_r = n$.

Take for example permutation $\sigma = 6\,5\,8\,7\,4\,1\,3\,2$. The $\ominus$-decomposition of σ is $\sigma = \ominus[6\,5\,8\,7, 4, 1\,3\,2]$. Furthermore σ has 2 RTL-minima which are $\sigma_6 = 1$ and $\sigma_8 = 2$.

Definition 3. *We denote $\sigma^{(i)} = \{\sigma_j \mid j < k_i \text{ and } \sigma_j > \sigma_{k_i}\}$ the restriction of σ to elements in the upper left quadrant of the i^{th} right-to-left (RTL) minima σ_{k_i}. The $\ominus_i$-decomposition of σ is the $\ominus$-decomposition of $\sigma^{(i)} = \ominus[B_1^{(i)}, \dots, B_{s_i}^{(i)}]$. In the sequel s_i always denote the number of blocks of $\sigma^{(i)}$ and $B_j^{(i)}$ the j^{th} block in the $\ominus_i$-decomposition.*

There are two key ideas in this article. First, among all possible sorting words for a 2-stack sortable permutation, there always exists a sorting word respecting some condition denoted $\mathcal{P}$. More precisely we prove that if a permutation σ is sortable then there exists a sorting process in which the elements that lie in the stacks just before a right to left minima k_i enters the stacks are exactly the elements of $\sigma^{(i)}$. A formal definition is given in Definition 16.

The second idea is to encode the different sortings of a permutation respecting $\mathcal{P}$ by a sequence of graphs $\mathcal{G}^{(i)}$ in which each node represents a stack configuration of a block $B_j^{(i)}$ and edges gives compatibility between the configurations. The index i is taken from 1 to the number of right-to-left minima of the permutation.

Section 2 study general properties on two-stack sorting and states which elements can move at each moment of a sorting process. Section 3 introduces the sorting graph $\mathcal{G}^{(i)}$ which encode all the sortings of a permutation at a given time t_i and gives an algorithm to compute this graph iteratively for all i from 1 to the number of right-to-left minima. Last section focusses on complexity analysis.

2 General results on two-stack sorting

2.1 Basic results

We saw that a sorting process can be described as a word on the alphabet $\{\rho, \lambda, \mu\}$. However not all words on the alphabet $\{\rho, \lambda, \mu\}$ describe sorting processes.

Definition 4 (stack word and sorting word)**.** *Let $\alpha \in \{\rho, \lambda, \mu\}$ and w a word on the alphabet $\{\rho, \lambda, \mu\}$. Then $|w|_\alpha$ denotes the number of letters α in w.*

A stack word *is a word $w \in \{\rho, \lambda, \mu\}^*$ such that for any prefix v of w, $|v|_\rho \geq |v|_\lambda \geq |v|_\mu$.*

A sorting word *is a stack word w such that $|w|_\rho = |w|_\lambda = |w|_\mu$.*

For any permutation σ, a sorting word for σ *is a sorting word encoding a sorting process with σ as input (leading to the identity of size $|\sigma|$ as output).*

Intuitively, stack words are words describing some operations ρ, λ, μ starting with empty stacks and an arbitrarily long input and they may be some elements in the stacks at the end of these operations, whereas sorting words are words encoding a complete sorting process (stacks are empty at the beginning and at the end of the process).

Definition 5 (subword)**.** *Let I be a set of integers.*

For any decorated word u we define $u_{|I}$ as the subword of u made of letters ρ_i, λ_i, μ_i with $i \in I$. For example, if $u = \rho_3\mu_5\lambda_3\rho_6\rho_7\lambda_6$ then $u_{|\{5,6\}} = \mu_5\rho_6\lambda_6$.

We extend this definition to stack words: given a permutation σ and a stack word w, $w_{|I}$ is the word of $\{\rho, \lambda, \mu\}^*$ obtaining from $\hat{w}^{\sigma}_{|I}$ by deleting indices from letters ρ_i, λ_i, μ_i.

Intuitively, $w_{|I}$ is the subword of w made of the operations of w that act on integers of I

Lemma 6. *For any stack word (resp. sorting word) w, $w_{|I}$ is also a stack word (resp. sorting word).*

Proof. As w is a stack word, for all i from 1 to $|\sigma|$, ρ_i appears before λ_i which itself appears before μ_i in $\hat{w}^{\sigma}_{|I}$. Therefore for any prefix v of $w_{|I}$, $|v|_\rho \geq |v|_\lambda \geq |v|_\mu$. If moreover w is a sorting word, let $\alpha \in \{\rho, \lambda, \mu\}$, then for any letter α_i in $\hat{w}^{\sigma}_{|I}$, ρ_i, λ_i and μ_i appear each exactly once in $\hat{w}^{\sigma}_{|I}$ thus $|w_{|I}|_\rho = |w_{|I}|_\lambda = |w_{|I}|_\mu$. □

Now we turn to stack configurations, beginning with linking stack words to stack configurations.

Definition 7 (Action of a stack word on a permutation)**.** *Let w be a stack word. Starting with a permutation σ as input, the stack configuration reached after performing operations described by the word w is denoted $c_\sigma(w)$. A stack configuration c is* reachable *for σ if there exists a stack word w such that $c = c_\sigma(w)$. In other words a stack configuration is reachable for σ if there exists a sequence of operations ρ, λ, μ leading to this configuration with σ as input.*

Lemma 8. *If $\sigma = \ominus[B_1, \dots B_k]$ then in any poppable stack configuration c reachable for σ, elements of B_i are below elements of B_j in the stacks for all $i < j$ (see Figure 2).*

Proof. Notice that by definition of a stack, elements of H are in increasing order from bottom to top for the indices. Moreover elements of V are in decreasing order from bottom to top for their value since from Lemma 2 they avoid pattern 12. This leads to the claimed property. □

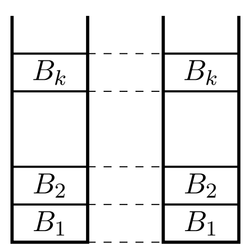
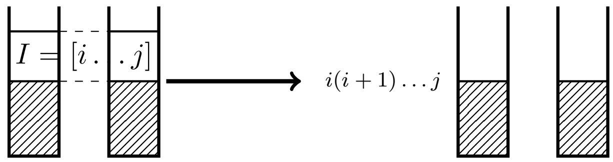


Figure 2: Poppable stack configuration reachable for $\ominus[B_1, \dots B_k]$.

Lemma 9. *Let σ be a 2-stack sortable permutation and $w = uv$ be a sorting word for σ. Assume that after performing operations of u, elements $1 \dots i-1$ have been output and elements $i \dots j$ are at the top of the stacks. Then there exists a sorting word $w' = uu'u''$ for σ such that u' consists only in moving elements $i \dots j$ from the stacks to the output in increasing order without moving any other elements.*



Proof. We claim that $u' = v_{|[i\ldots j]}$ and $u'' = v_{|![i\ldots j]}$ satisfy the desired property, where $![i\ldots j]$ is the set of integers $[1\ldots|\sigma|]\setminus[i\ldots j]$. This can be checked using decorated words associated to w and w' and noticing that $v_{|[i\ldots j]} = out_{c_\sigma(u)}([i\ldots j])$ and $v_{|![i\ldots j]} = v_{|>j}$ since by hypothesis after performing operations of u, elements $1\ldots i-1$ have been output and elements $i\ldots j$ are at the top of the stacks. $\square$

The stack configurations for a sorting process encode the elements that are currently in the stacks. But some elements are still waiting in the input and some elements have been output. To fully characterize a configuration, we define an *extended* stack configuration of a permutation σ of size n to be a pair (c,i) where $i \in \{1,\ldots n+1\}$ and c is a poppable stack configuration made of all elements within $\sigma_1,\sigma_2,\ldots,\sigma_{i-1}$ that are greater than a value p. The elements $\sigma_i,\ldots,\sigma_n$ are waiting to be pushed and elements $\sigma_j < p, j < i$ have already been output. Notice that we don't need the configuration to be reachable.

Definition 10. *Let σ be a permutation and (c,i) be an extended stack configuration of σ. Then the extended stack configuration (c',j) of σ is* accessible *from (c,i) if the stack configuration (c',j) can be reached starting from (c,i) and performing operations ρ,λ and μ such that moves μ perfomed output elements of $c\cup\{\sigma_i\ldots\sigma_n\}$ in increasing order.*

For example, if $\sigma = 2\,3\,1\,6\,5\,8\,4\,7$ then $(\begin{array}{|c|}8\\\hline\end{array}\ \begin{array}{|c|}5\\6\\\hline\end{array}, 7)$ is accessible from $(\begin{array}{|c|}2\\3\\\hline\end{array}\ \sqcup, 4)$ by the sequence of operations $\mu_2\mu_3\rho_6\rho_5\rho_8\lambda_8$. But $(\begin{array}{|c|}2\\3\\\hline\end{array}\ \begin{array}{|c|}6\\\hline\end{array}, 5)$ is not accessible from $(\begin{array}{|c|}1\\2\\\hline\end{array}\ \begin{array}{|c|}3\\\hline\end{array}, 4)$.

Indeed notice that the question of whether a permutation is 2-stack sortable can be reformulated as :

Is $(\sqcup\ \sqcup, n+1)$ **accessible from** $(\sqcup\ \sqcup, 1)$ **?**

To solve this problem is the main goal of this article and is somehow hard, however some special cases are easier to deal with. The following Lemma give conditions on the involved configurations under which the compatibility decision problem is linear and can by solved by the *isAccessible* procedure given in Algorithm 1. In the last sections, we show how more general cases can be solved using this Lemma.

Lemma 11. *Let σ be a permutation of size n and (c,i), (c',j) two extended stack configurations of σ with $i<j$. Let E be the set of elements of c and F those of c'.*

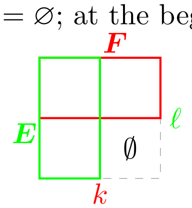
- *If there exists $k,\ell \in \{1\ldots n\}$ such that $E = \{\sigma_m \mid m \le k\}$ and $F = \{\sigma_m \mid \sigma_m \ge \ell\}$*
- *If moreover $E \cup F = \sigma$*

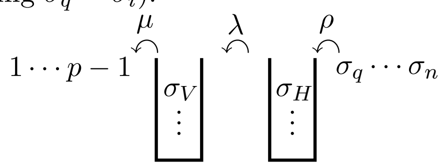
Then we can decide in linear time whether (c',j) is accessible from (c,i) using Algorithm 1.

Proof. We prove by case study that there is no choice between operations ρ,λ,μ at each time step. This is illustrated by Algorithm 1. We first prove its correctness before studying its complexity.

We start with configuration $curr = c$. By studying specific elements of the current configuration $curr$, we prove that we can always decide which operation should be performed to transform $curr$ into c'. If at any step this operation is forbidden then c' is not accessible from $curr$. Thus repeating the following process will eventually lead to decide whether c' is accessible from c.

Notice that by definition, c and c' are poppable thus $curr$ has to be poppable, hence to avoid the three patterns of Lemma 2. Let p be the next element to be output, *i.e.* the smallest element of $c \cup \{\sigma_i \ldots \sigma_n\}$. Let σ_H (resp. σ_V) be the topmost element of H (resp. of V) and σ_q be the element waiting in the input to be pushed onto H (σ_q may not exist and in that case $\sigma_q = \varnothing$; at the beginning $\sigma_q = \sigma_i$).





- If $\sigma_V = p$ then we perform μ thanks to Lemma 9.
- Otherwise operation μ is forbidden. We have to chose between ρ and λ. Moreover $p \notin V$ as V is in decreasing order from bottom to top.
  1. Suppose that $\sigma_H < \ell$. This means that $\sigma_H \notin F$ *i.e.* $\sigma_H \notin c'$. Notice that by definition of p, $p \leq \sigma_H$ thus $p \notin c'$. Moreover $p \notin V$ thus $p \in H$. If $p = \sigma_H$ then, by Lemma 9, we can pop out p. Thus we perform λ. If $\sigma_H \neq p$, then we will prove that all elements x such that $p \leq x \leq \sigma_H$ form an interval at the top of the stacks. Those elements are all in the stacks by definition of ℓ and p. As V is decreasing, the elements of $[p \ldots \sigma_H]$ belonging to V are at the top of it. Consider now the position of those elements in H.

     Suppose that it is not an interval. Then it exists an element x in H such that $x < \sigma_H$ and there is an element $y > \sigma_H$ between x and σ_H. But in that case, elements $xy\sigma_H$ form the pattern 1 3 2 and $curr$ is not poppable so any movement is allowed here ρ, λ or μ because we will never reach c'.

     Suppose now that the elements $[p \ldots \sigma_H]$ form an interval in H and V. Then as $p \in H$ is the smallest element, by Lemma 9, we want to pop out elements $[p \ldots \sigma_H]$, hence we perform λ.

     In conclusion, if $\sigma_H < \ell$ we perform λ.
  2. If not, then $\sigma_H \geq \ell$ and thus $\sigma_H \in c'$. Once again there are different cases:
     (a) If $H = \varnothing$ then λ is forbidden, thus we perform ρ.
     (b) If $\sigma_H \in H(c')$, it must stay in H thus λ is forbidden and we perform ρ.
     (c) Else $\sigma_H \in V(c')$.
        – If $\sigma_q \in H(c')$ then ρ is forbidden because σ_q would prevent σ_H from moving. Thus we perform λ.
        – Else $\sigma_q \in V(c')$. If $\sigma_H > \sigma_q$, as $\sigma_H \in V(c')$, ρ is forbidden otherwise we cannot put σ_q above σ_H in V. Thus we perform λ.
        – Otherwise $\sigma_H, \sigma_q \in V(c')$ and $\sigma_H < \sigma_q$. λ is forbidden otherwise we cannot put σ_H above σ_q in V. Thus we perform ρ.

We have proved that at each step of the algorithm, we know which move we have to do if we want to reach c'. Moreover while $q < j$ or $p < \ell$ or $\sigma_H \in V(c')$, it is impossible that $curr = c'$ so we have to continue. Conversely if $q \geq j$ and $p \geq \ell$ and $\sigma_H \notin V(c')$ then ρ and μ and λ are forbidden and we have to stop. Then if $curr = c'$, c' is accessible from c, otherwise c' is not accessible from c.

Finally there are at most $3n$ steps since at each step of the algorithm we perfom a move ρ, λ or μ. Moreover each step takes a constant time, therefore the algorithm runs in linear time. □

Algorithm 1: isAccessible$\big((c, i), (c', j), \sigma\big)$

```
Data: σ a permutation and (c,i),(c',j) two stack configurations of σ respecting
      conditions of Lemma 11
Result: true or false depending whether the configuration c' is accessible from c
begin
    Put configuration c in the stacks H and V;
    p ← the smallest element of c ∪ {σ_i ... σ_n} (next element to be output);
    q ← i (next index of σ that must enter the stacks);
    while q < j OR p < ℓ OR σ_H ∈ V(c') do
        if σ_V = p then
            Perform μ; p ← p + 1;
        else
            if σ_H < ℓ then
                Perform λ;
            else
                if H = ∅ OR σ_H ∈ H(c') then
                    Perform ρ; q ← q + 1;
                else
                    if σ_q ∈ H(c') OR σ_H > σ_q then
                        Perform λ;
                    else
                        Perform ρ; q ← q + 1;
    Return (H, V) == c';
```

In the sequel of this article, we do not compute all possible stack configurations during a sorting process of a given permutation σ but indeed focus on specific steps of the sorting. We study the possible stack configurations at each time step t_i corresponding to the moment just before the right to left minimum σ_{k_i} is pushed onto stack H. Those configurations are configurations (c, k_i) accessible from $(\sqcup \bigsqcup, 1)$.

We will prove that we can add two different restrictions on these configurations. First, (c, k_i) must be a pushall stack configuration of $\sigma^{(i)}$ (see below). Second (c, k_i) must be an evolution of some configuration (c', k_{i-1}) between time t_{i-1} and t_i.

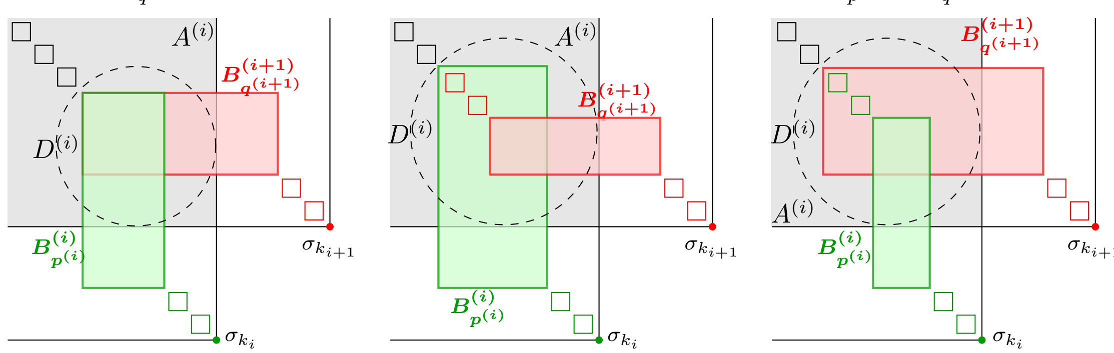
Definition 12 (pushall configuration)**.** *A stack configuration is a* pushall *stack configuration of σ if it is poppable, total and reachable for σ.*

2.2 From time t_i to time t_{i+1}

Thanks to the previous decomposition into different time steps corresponding to each moment a right-to-left minima is pushed onto H and our previous work [3] on 2-stack pushall sortable permutations, we can give a polynomial algorithm deciding whether a permutation is 2-stack sortable. Indeed, we will prove that it is enough to consider configurations such that for each t_i the only elements in the stacks are exactly those of $\sigma^{(i)}$. But $\sigma^{(i)}$ is a permutation that ends with its smallest element such that a sorting consists in pushing all elements into the stacks then popping all elements out. Those possibilities are described in [3] where Proposition 4.8 gives all possible pushall stack configurations. When a permutation is $\ominus$-indecomposable, Theorem 4.4 of [3] states that the number of possible pushall stack configurations is linear in the size of the permutation. This will ensure that our algorithm runs in polynomial time. Using this result, we now have the possible total stack configurations at time t_1.

The key idea for computing possible stack configurations at time t_i relies on Lemma 15. Informally, it is possible to decide whether a configuration at time t_i can evolved into a specific configuration at time t_{i+1}. Moreover, during this transition, only a few moves are undetermined. Indeed the largest elements won't move, the smallest one will be pushed accordingly to [3] and the remaining ones form a $\ominus$-indecomposable permutation that will allow us to exhibit a polynomial algorithm.

First of all we denote by $A^{(i)}$ the common part of the permutations $\sigma^{(i)}$ and $\sigma^{(i+1)}$, that is, $A^{(i)} = \sigma^{(i)} \bigcap \sigma^{(i+1)} = \{\sigma_j \mid j < k_i \text{ and } \sigma_j > \sigma_{k_{i+1}}\}$. This subpermutation $A^{(i)}$ intersects $\ominus$-indecomposable blocks of $\sigma^{(i)}$ and $\sigma^{(i+1)}$. Let $p^{(i)}$ (resp. $q^{(i+1)}$) be the index such that $B^{(i)}_{p^{(i)}}$ (resp. $B^{(i+1)}_{q^{(i+1)}}$) contains the smallest value of $A^{(i)}$. Let $D^{(i)} = (B^{(i)}_{p^{(i)}} \bigcup B^{(i+1)}_{q^{(i+1)}}) \bigcap A^{(i)}$.



Lemma 13. *For any $j < \min(p^{(i)}, q^{(i+1)})$, $B^{(i)}_j = B^{(i+1)}_j$.*

Lemma 14. *Let $\sigma_\ell \in A^{(i)}$. During a sorting process of σ, elements σ_m such that $\sigma_m > \sigma_\ell$ and $m < \ell$ do not move between t_i and t_{i+1}.*

Proof. Let σ_m be an element such that $m < \ell$ and $\sigma_m > \sigma_\ell$. As $\sigma_\ell \in A^{(i)}$, $\sigma_\ell > \sigma_{k_{i+1}}$ and $j < k_i$, so does $\sigma_m > \sigma_{k_{i+1}}$ and $m < k_i$. Hence both elements σ_m, σ_ℓ lie in the stacks between t_i and t_{i+1} (they cannot be output as $\sigma_{k_{i+1}}$ must be output first). Suppose that σ_m is in H at time t_i. As $m < \ell$, element σ_ℓ is pushed after σ_m into the stacks, thus either σ_ℓ is above σ_m in H or lies in V at time t_i and t_{i+1}. So, σ_m cannot move into V, otherwise σ_ℓ would be under it in V and V would contain a pattern 12. So, σ_m stay in H.

Suppose now that σ_m is in V at time t_i. As noticed previously, this element is not output at time t_{i+1}. So it also lies in stack V at time t_{i+1}, proving the lemma. $\square$

In the following we study conditions for 2 total pushall stack configurations c and c' corresponding to stack configuration of $\sigma^{(i)}$ and $\sigma^{(i+1)}$ to be accessible one from the other, that is, if we can move elements starting from c at time t_i to obtain c' at time t_{i+1}.

Lemma 15. *Let (c, k_i) (resp. (c', k_{i+1})) be a total stack configuration of $\sigma^{(i)}$ (resp. $\sigma^{(i+1)}$). Let $\pi = \sigma_{|B^{(i)}_{p^{(i)}} \bigcup B^{(i+1)}_{q^{(i+1)}}}$ then (c', k_{i+1}) is accessible from (c, k_i) for σ iff:*

1. *$(c'_{|\pi}, |\pi| + 1)$ is accessible from $(c_{|\pi}, \sharp(D^{(i)} \bigcup B^{(i)}_{p^{(i)}}) + 1)$ for π.*
2. *$\forall j < min(p^{(i)}, q^{(i+1)})$, $c_{|B^{(i)}_j} = c'_{|B^{(i)}_j}$.*
3. *$\forall j > q^{(i+1)}, c'_{|B^{(i+1)}_j}$ is a reachable configuration.*

Proof. Suppose first that (c', k_{i+1}) is accessible from (c, k_i). This means that we can go from c to c' using operations represented by the decorated word $\hat{w}$. These operations are stable that is for all I, $c'_{|I}$ is accessible from $c_{|I}$. To do so, we just extract operations corresponding to elements of I. Indeed the decorated word $\hat{w}_{|I}$ allow to transform c into c'. This proves the first point of Lemma 15.

Let $\sigma_\ell \in B^{(i)}_p$. Lemma 14 ensures that elements of $B^{(i)}_j$ with $j < p^{(i)}$ do not move between t_i and t_{i+1} proving the second point of Lemma 15.

Finally, elements of $B^{(i+1)}_j$ for $j > q^{(i+1)}$ are pushed iteratively when going from c to c'. Those elements stay in the stacks as $\sigma_{k_{i+1}}$ which is smaller is pushed after them. Thus they correspond to a pushall configuration.

Conversely, suppose that we have the 3 different points above, we must prove that (c', k_{i+1}) is accessible from (c, k_i) for σ. We start by taking the stack configuration c and we will prove that we can obtain c' by moving elements. First of all, as c is a pushall stack configuration, and as elements of B_ℓ for $\ell > p$ are the smallest one and have been pushed last into the stacks they are at the top of the stacks (see Lemma 8). Thus we can pop them and output them in increasing order using Lemma 9.

The remaining elements in the stacks don't move in the preceding operation, thus stay in the same position than in c. In that configuration, elements of $B^{(i)}_{p^{(i)}}$ are the smallest ones and have been pushed the latter in the stacks. Hence they lie at the top of the stacks.

Then using point 1 of our hypothesis, we can move those elements together with pushing elements of $B^{(i+1)}_{q^{(i+1)}} \setminus B^{(i)}_{p^{(i)}}$ so that all those elements (that is elements of π) are in the same position than in c'. Then, by hypothesis item 3, $\forall j > q^{(i+1)}, c'_{|B^{(i+1)}_j}$ is a reachable configuration. Thus we can push its elements into the stacks in the same relative order than in c' (see Lemma 8). During these operations we ensure that elements of B_ℓ with $\ell \geq min(p^{(i)}, q^{(i+1)})$, $c_{|B^{(i)}_j}$ are in the same position in our configuration than in c'. Point 2 ensures that we indeed obtain c'. ☐

The preceding Lemma describes exactly which elements can move between t_i and t_{i+1} and how they move. But the hypothesis of Lemma 15 are restrictive that is configurations c and c' must be two total stack configurations of $\sigma^{(i)}$ and $\sigma^{(i+1)}$. Thus, we first prove that among all sortings of a 2-stack sortable permutation, there exists at least one for which the stack configurations at time t_i contains exactly the elements of $\sigma^{(i)}$ for all i.

Definition 16 (Properties (P_i) and (P))**.** *Let σ be a permutation and w a sorting word for σ. w verifies (P_i) if and only if*

(i) $\rho_{\sigma_{k_i}}\lambda_{\sigma_{k_i}}\mu_{\sigma_{k_i}}$ is a factor of w.

(ii) μ_{σ_j} appears before $\rho_{\sigma_{k_i}}$ for all $\sigma_j < \sigma_{k_i}$.

(iii) All operations μ_{σ_ℓ} with $\sigma_\ell \in B_j^{(i)}$ and $j \in [p^{(i)}+1..s_i]$ appear before $\rho_{\sigma_{k_{i+1}}}$ in w.

where σ_{k_i} is the i^{th} right to left minima of the permutation and $\sigma^{(i)} = \ominus[B_1^{(i)}, \ldots, B_{s_i}^{(i)}]$.

If a word w verifies Property (P_i) for all i then we say that w verifies Property (P).

Lemma 17. *If the sorting word encoding a sorting process of σ verifies Property (P_i), then at time t_i the elements currently in the stacks are exactly those of $\sigma^{(i)}$.*

Proof. By definition of time t_i (just before σ_{k_i} enters the stacks) each element in the stacks has an index smaller than k_i. Moreover among elements of index smaller than k_i, those of value greater than σ_{k_i} cannot have been output by definition of a sorting, and those of value smaller than σ_{k_i} have already been output since w satisfies item *(ii)* of Property (P_i). $\square$

Lemma 18. *Let w be a sorting word for a permutation σ, r be the number of RTL-minima of σ and $\ell \in [1..r]$. If w verifies (P_i) for $i \in [1..\ell-1]$ then there exists a sorting word w' for σ that verifies (P_i) for $i \in [1..\ell]$.*

Proof. Consider the sorting process of σ encoded by w. The key idea is to prove that the smallest elements are at the top of the stacks so that we can transform the word w thanks to Lemma 9.

Property (ii) for (P_ℓ) states that μ_{σ_j} should appear before $\rho_{\sigma_{k_\ell}}$ for all $\sigma_j < \sigma_{k_\ell}$. Suppose that there still exists an element σ_j with $\sigma_j < \sigma_{k_\ell}$ in the stacks just before σ_{k_ℓ} is pushed into the stacks. We prove that this element can be popped out before σ_{k_ℓ} is pushed. Let σ_{j_0} be the smallest element still in the stacks just before $\rho_{\sigma_{k_i}}$. By definition, elements smaller than σ_{j_0} have already been output. Consider interval $I = [\sigma_{j_0}, \sigma_{k_\ell} - 1]$. Those elements are still in the stacks. If they are at the top of the stacks they can be output using Lemma 9. If not, there exists in the stacks an element $x \notin I$ above an element $y \in I$. As V is decreasing, those elements are in H. Moreover $x > \sigma_{k_\ell} > y$. Then σ_{k_ℓ} cannot be pushed as it will create a pattern 132 in H with elements x and y. Thus I is at the top of the stacks and we can output it before σ_{k_ℓ} is pushed onto H: using Lemma 9, we build from w a sorting word $w^{(1)}$ for σ satisfying (P_i) for $i \in [1..\ell-1]$ and Property (ii) of (P_ℓ). This means that $w^{(1)}$ can be decomposed as $w^{(1)} = u\rho_{\sigma_{k_\ell}}v$ such that the stack configuration $c_\sigma(u)$ respects the following constraint: elements $1, \ldots, \sigma_{k_\ell}$ are not in the stacks.

So if we consider the stack configuration $c_\sigma(u\rho_{\sigma_{k_\ell}})$, element σ_{k_ℓ} is at the top of H and since $out_{c_\sigma(u\rho_{\sigma_{k_\ell}})}(\sigma_{k_\ell}) = \lambda_{\sigma_{k_\ell}}\mu_{\sigma_{k_\ell}}$ we can use Lemma 9 to change the sorting word $w^{(1)}$ into a sorting word $w^{(2)} = u\rho_{\sigma_{k_\ell}}\lambda_{\sigma_{k_\ell}}\mu_{\sigma_{k_\ell}}v'$, satisfying Property (i) for (P_ℓ).

Now we show considering the stack configuration $c = c_\sigma(u\rho_{\sigma_{k_\ell}}\lambda_{\sigma_{k_\ell}}\mu_{\sigma_{k_\ell}})$ how to transform the word $w^{(2)}$ into a word $w' = u\rho_{\sigma_{k_\ell}}\lambda_{\sigma_{k_\ell}}\mu_{\sigma_{k_\ell}}v^{(1)}v^{(2)}$ with $v^{(1)} = out_c(B_{p^{(\ell)}+1}^{(\ell)} \cup \cdots \cup B_{s_\ell}^{(\ell)})$. This will conclude the proof.

Notice that elements of c are exactly those of $\sigma^{(\ell)}$ since the last operations performed are $\rho_{\sigma_{k_\ell}}\lambda_{\sigma_{k_\ell}}\mu_{\sigma_{k_\ell}}$ and elements are pushed in the stacks in increasing order of indices and output in increasing order of values. Thus $out_c(B^{(\ell)}_{p^{(\ell)}+1} \cup \cdots \cup B^{(\ell)}_{s_\ell}) = out(B^{(\ell)}_{s_\ell}) \ldots out(B^{(\ell)}_{p^{(\ell)}+1})$ (see Lemma 8). We show by induction on j from s_ℓ to $p^{(i)}+1$ that we can build a sorting word for σ of the form $u\rho_{\sigma_{k_\ell}}\lambda_{\sigma_{k_\ell}}\mu_{\sigma_{k_\ell}}v^{(1,j)}v^{(2,j)}$ with $v^{(1,j)} = out(B^{(\ell)}_{s_\ell}) \ldots out(B^{(\ell)}_j)$. For $j = s_\ell$ that is a word in which elements of block B_{s_ℓ} are output immediately after σ_{k_ℓ} has been output. By definition of s_ℓ and because elements of c are exactly those of $\sigma^{(\ell)}$, all elements of B_{s_ℓ} lie in the stacks in configuration c, are the smallest elements in this configuration and lie at the top of the stacks in configuration c (see Lemma 8). Hence, using Lemma 9, there exist a sorting word $w^{(3)}$ for σ such that $w^{(3)} = u\rho_{\sigma_{k_\ell}}\lambda_{\sigma_{k_\ell}}\mu_{\sigma_{k_\ell}}out(B_{s_\ell})v''$. Repeating this operation for all blocks B_j with j from $s_\ell - 1$ to $p^{(i)}+1$, we have Property (iii). □

Notice that Property (P_0) is an empty property satisfied by any sorting word. Using recursively Lemma 18 we can transform any sorting word into a sorting word satisfying Property (P), leading with Lemma 17 to the following theorem:

Theorem 19. *If σ is 2-stack sortable then there exists a sorting word of σ respecting Property (P). In particular, in the sorting process that this word encodes, the elements currently in the stacks at time t_i are exactly those of $\sigma^{(i)}$.*

Theorem 19 ensures that if a permutation is sortable then there exists a sorting in which at each time step t_i, elements in the stacks are exactly those of $\sigma^{(i)}$. Thus stack configurations at time t_i and t_{i+1} satisfy hypothesis of Lemma 15 and we can apply it to decide if a permutation if 2-stack sortable.

3 An iterative algorithm

3.1 A fisrt naïve algorithm

From Theorem 19 a permutation σ is 2-stack sortable if and only if it admits a sorting process satisfying Property (P). The main idea is to compute the set of sorting processes of σ satisfying Property (P) and decide whether σ is 2-stack sortable by testing its emptiness.

Verifying (P) means verifying (P_j) for all j from 1 to r, r being the number of right-to-left minima (whose indices are denoted k_j). The algorithm proceeds in r steps: for all i from 1 to r we iteratively compute the sorting processes of $\sigma_{\leq k_i}$ verifying (P_ℓ) for all ℓ from 1 to i. As $\sigma_{\leq k_r} = \sigma$, the last step gives sorting processes of σ satisfying Property (P).

By "compute the sorting processes of $\sigma_{\leq k_i}$" we mean compute the stack configuration just before σ_{k_i} enters the stacks in such a sorting process. Note that this is also the stack configuration just after σ_{k_i} has been output since $\rho_{\sigma_{k_i}}\lambda_{\sigma_{k_i}}\mu_{\sigma_{k_i}}$ is a factor of any word verifying (P).

Definition 20. *We call P_i-stack configuration of σ a stack configuration $c_\sigma(w)$ for which there exists u such that the first letter of u is $\rho_{\sigma_{k_i}}$ and wu is a sorting word of $\sigma_{\leq k_i}$ verifying (P) for $\sigma_{\leq k_i}$ (that is, verifying (P_ℓ) for all ℓ from 1 to i).*

Lemma 21. *For any i from 1 to r, $\sigma_{\leq k_i}$ is 2-stack sortable if and only if the set of P_i-stack configurations of σ is nonempty. In particular, σ is 2-stack sortable if and only if the set of P_r-stack configurations of σ is nonempty.*

Proof. This is a direct consequence of Definition 20 and Theorem 19. □

Lemma 22. *Any P_i-stack configuration of σ is a pushall stack configuration of $\sigma^{(i)}$ accessible from some P_{i-1}-stack configurations of σ.*

Proof. By definition of (P), each P_i-stack configurations of σ is accessible from some P_{i-1}-stack configurations of σ (take the prefix of w that ends just before $\rho_{\sigma_{k_{i-1}}}$). Moreover it is a pushall stack configuration of $\sigma^{(i)}$ from Lemma 17. $\square$

As explained above, the algorithm proceeds in r steps such that after step i we know every P_i-stack configuration of σ and we want to compute the P_{i+1}-stack configurations of σ at step $i+1$. As configurations for $i+1$ are a subset of pushall stack configurations of $\sigma^{(i+1)}$, a possible algorithm is to take every pair of configurations (c, c') with c being a P_i-stack configuration of σ (computed at step i) and c' be any pushall stack configuration of $\sigma^{(i+1)}$ (given by Algorithm 5 of [3]). Then we can use Algorithm 1 to decide whether c' is accessible from c for σ. This leads to the following algorithm deciding whether a permutation σ is 2-stack sortable:

Algorithm 2: *isSortableNaive*

```
Data: σ a permutation
Result: true or false depending whether σ is 2-stack sortable
begin
    E, F two empty sets;
    E ← PushallConfigs(σ^(1));
    for i from 2 to r do
        F ← ∅;
        for c in E do
            for c' in PushallConfigs(σ^(i)) do
                if isAccessible((c, k_i), (c', k_{i+1}), σ) then
                    F ← F ∪ c';
        E ← F;
    if E is empty then
        return false;
    else
        return true;
```

Notice that at step i, the set E computed contains all P_i-stack configurations of σ but may contain some other configurations. However since each configuration of E is a pushall configuration of $\sigma^{(i)}$ and is accessible for σ from some pushall configurations of $\sigma^{(i-1)}$, each configuration of E indeed corresponds to some sorting procedure of $\sigma_{\leq k_i}$, proving the correctness of Algorithm 2.

But this algorithm is not polynomial. Indeed the number of P_i-stack configurations of σ is possibly exponential. However this set can be described by a polynomial representation as a graph $\mathcal{G}^{(i)}$ and we can adapt Algorithm 2 to obtain a polynomial algorithm. In this adapted algorithm, the set E computed at step i is exactly the set of P_i-stack configurations of σ.

3.2 Towards the sorting graph

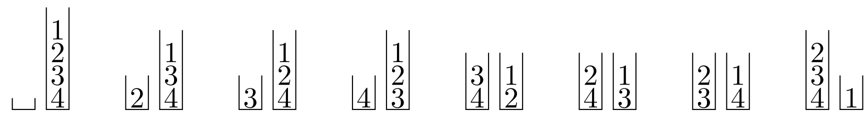
We now explain how to adapt Algorithm 2 to obtain a polynomial algorithm. Instead of computing all P_i-stack configurations of σ (which are pushall stack configurations of $\sigma^{(i)}$), we compute the restriction of such configurations to blocks $B_j^{(i)}$ of the $\ominus$-decomposition of $\sigma^{(i)}$. By Lemma 8, those configurations are stacked one upon the others. The stack configurations of any block $B_j^{(i)}$ are labeled with an integer which is assigned when the configuration is computed. Those pairs configurations / integer will be the vertices of the graph $\mathcal{G}^{(i)}$ which we call a *sorting graph*, the edges of which representing the configurations that can be stacked one upon the other. Vertices of the graph $\mathcal{G}^{(i)}$ are partitioned into levels corresponding to blocks $B_j^{(i)}$. To ensure the polynomiality of the representation, we will prove that a given integer label could only appear once per level of the graph $\mathcal{G}^{(i)}$. As those numbers are assigned to configurations when they are created, each integer corresponding to a pushall stack configuration, from [3] there exists only a polynomial number of disctincts integers thus of vertices. This will be explained in details in the next section. The integer indeed can be seen as the memory of the configuration that encodes its history since it has been created: two configurations which have the same label come from the same initial pushall configuration.

More precisely a sorting graph $\mathcal{G}^{(i)}$ for a permutation σ of size n and an index i verifies the following properties:

- Vertices of $\mathcal{G}^{(i)}$ are partitioned into s_i subsets $V_j^{(i)}$ with $j \in [1 \dots s_i]$.
- For any $j \in [1 \dots s_i], |V_j^{(i)}| \leq 9n + 2$.
- Each vertex $v \in \mathcal{G}^{(i)}$ is a pair (c, ℓ) with c a stack configuration and ℓ an index called *configuration index*.
- All configuration indices are distinct inside a graph level $V_j^{(i)}$
- $(c, \ell) \in V_j^{(i)} \Rightarrow c$ is a pushall stack configuration of $B_j^{(i)}$ accessible for σ.
- There are edges only between adjacent blocks $V_j^{(i)}$, $V_{j+1}^{(i)}$.
- Paths between vertices of $V_1^{(i)}$ and $V_{s_i}^{(i)}$ corresponds to stack configurations of $\sigma^{(i)}$. More precisely such paths are in one-to-one correspondence with P_i-stack configurations of σ (that is, stack configurations corresponding to a sorting of $\sigma_{\leq k_i}$ respecting (P) just before σ_{k_i} is pushed onto H).
- For any vertex v of $\mathcal{G}^{(i)}$, there is a path between vertices of $V_1^{(i)}$ and $V_{s_i}^{(i)}$ going through v.

Though the definition of sorting graph is complex, its use will be quite understandable and easy. Look for example at the permutation $\sigma = 4321$. There is only one right to left minimum which is 1. Compute all possible stack configurations just after 1 enters H. At this time, all elements are in the stacks since the first element which must be output is 1. More formally, we are looking at the pushall stack configurations of σ with 1 in H.

There are 8 different such configurations which are:



The $\ominus$-decomposition of σ is $\sigma = \ominus[4,3,2,1]$. We build a graph with 4 levels, each level corresponding to pushall stack configurations of a block.

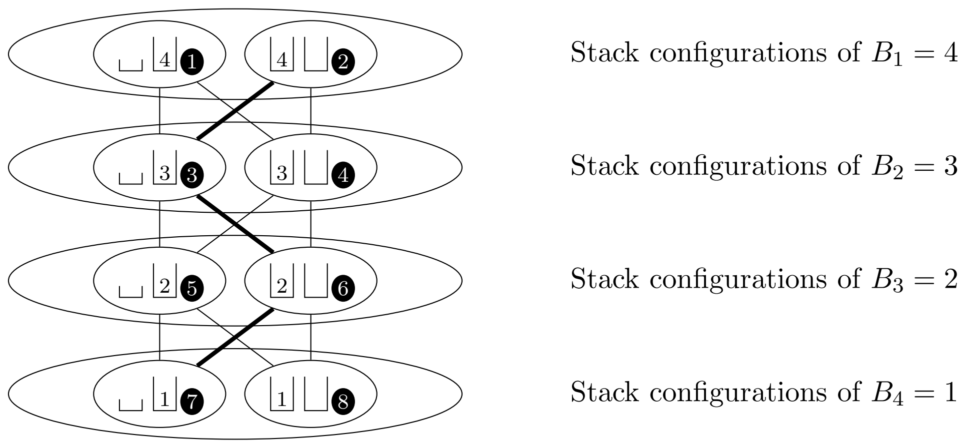
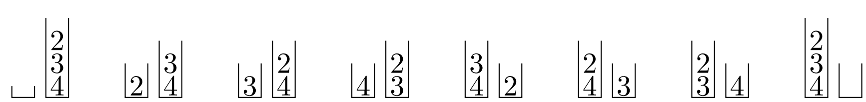


Figure 3: Graph encoding pushall stack configurations of $\sigma = 4321$.

Then the 8 configurations are found taking each of the 8 different paths going from any configuration of B_1 to configuration $\sqcup\,\boxed{1}$ of B_4. In Figure 3, the thick path gives the stack configuration $\begin{array}{|c|}2\\4\\\hline\end{array}\,\begin{array}{|c|}1\\3\\\hline\end{array}$ by stacking the selected configuration of B_4 above the configuration of B_3 and so on.

But in the last level B_4 we only consider configuration $\sqcup\,\boxed{1}$ so this level is useless. The sorting graph $\mathcal{G}^{(1)}$ for $\sigma = 4321$ encodes pushall stack configurations of $\sigma^{(1)} = 432$, corresponding to stack configurations just *before* 1 enters H (and not after as above).

There are 8 different such configurations which are:



As the $\ominus$-decomposition of $\sigma^{(1)}$ is $\sigma^{(1)} = \ominus[4,3,2]$, the sorting graph $\mathcal{G}^{(1)}$ has 3 levels.

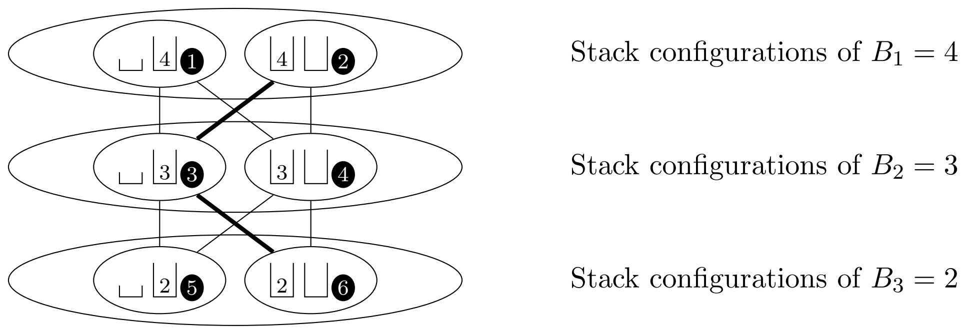


Figure 4: Sorting graph $\mathcal{G}^{(1)}$ of $\sigma = 4321$.

Then the 8 configurations are found taking each of the 8 different paths going from any configuration of B_1 to any configuration of B_3. In Figure 4, the thick path gives the stack configuration $\begin{array}{|c|}2\\4\end{array}\ \begin{array}{|c|}3\end{array}$ by stacking the selected configuration of B_3 above the configuration of B_2 and so on.

We transform Algorithm 2 to a polynomial algorithm by computing at step i not all P_i-stack configurations of σ, but instead the sorting graph $\mathcal{G}^{(i)}$ encoding them. The graph $\mathcal{G}^{(i)}$ is computed iteratively from the graph $\mathcal{G}^{(i-1)}$ for any i from 2 to r. The way $\mathcal{G}^{(i)}$ is computed from $\mathcal{G}^{(i-1)}$ depends on the relative values of $p^{(i)}$ and $q^{(i+1)}$. By definition of a sorting graph given p.14, if at any step $\mathcal{G}^{(i)}$ is empty, it means that $\sigma_{\leq k_i}$ is not sortable (from Theorem 19) and so is σ thus the algorithm returns `false`. This is summarized in Algorithm 3.

Algorithm 3: *isSortable*

Data: σ a permutation
Result: `true` or `false` depending whether σ is 2-stack sortable
begin
  $\mathcal{G} \leftarrow ComputeG1$;
  for i *from* 2 *to* r **do**
    if $p^{(i)} = q^{(i+1)}$ **then**
      $\mathcal{G} \leftarrow iteratepEqualsq(\mathcal{G})$ or **return** `false`
    else
      if $p^{(i)} < q^{(i+1)}$ **then**
        $\mathcal{G} \leftarrow iteratepLessThanq(\mathcal{G})$ or **return** `false`
      else
        $\mathcal{G} \leftarrow iteratepGreaterThanq(\mathcal{G})$ or **return** `false`
  return `true`

In the next subsections we describe the subprocedures used in our main algorithm $isSortable(\sigma)$.

3.3 First step: $\mathcal{G}^{(1)}$

In this subsection, we show how to compute the P_1-stack configurations of σ, that is, the stack configurations corresponding to time t_1 for sorting words of $\sigma_{\leq k_1}$ that respect (P) for $\sigma_{\leq k_1}$.

From Lemma 22, such a stack configuration is a pushall stack configuration of $\sigma^{(1)}$. Conversely since $\sigma_{k_1} = 1$, $\sigma^{(1)} = \sigma_{<k_1}$ and each sorting word of $\sigma_{\leq k_1}$ respects (P_1) for $\sigma_{\leq k_1}$. Thus the set of P_1-stack configurations of σ is the set of pushall stack configurations of $\sigma^{(1)}$.

By Proposition 4.7 of [3], these stack configurations are described by giving the set of stack configurations for each block of the $\ominus$-decomposition of $\sigma^{(1)}$. More precisely, with $\sigma^{(1)} = \ominus[B_1^{(1)}, \dots, B_{s_1}^{(1)}]$ there is a bijection from $pushallConfigs(B_1^{(1)}) \times \cdots \times pushallConfigs(B_{s_1}^{(1)})$ onto $pushallConfigs(\sigma^{(1)})$ by stacking configurations one upon the other (as in Lemma 8). As a consequence, from Lemma 21 $\sigma_{\leq k_1}$ is not sortable if and only if a set $pushallConfigs(B_j^{(1)})$ is empty.

Moreover it will be useful to label the configurations computed so that we attach a distinct integer to each stack configuration when computed.

At this point, we have encoded all configurations corresponding to words respecting P up to the factor $\rho_1\lambda_1\mu_1$.

The obtained graph is $\mathcal{G}^{(1)}$. This step is summarized in Algorithm 4.

Algorithm 4: ComputeG1

Data: σ a permutation, num a global integer variable
Result: `false` if $\sigma_{\leq k_1}$ is not sortable, the sorting graph $\mathcal{G}^{(1)}$ otherwise.
begin
  $E = \varnothing$;
  Compute $\sigma^{(1)}$ and its $\ominus$-decomposition $\ominus[B_1^{(1)}, \dots, B_{s_1}^{(1)}]$;
  for j *from* 1 *to* $s_1^{(1)}$ **do**
    $V_j^{(1)} \leftarrow \varnothing$;
    $S = pushallConfigs(B_j^{(1)})$;
    if $S = \varnothing$ **then**
      return `false`;
    else
      for $s \in S$ **do**
        $V_j^{(1)} \leftarrow V_j^{(1)} \bigcup \{(s, num)\}$;
        $num \leftarrow num + 1$;
      if $j > 1$ **then**
        $E = E \bigcup \{(s, s'), s \in V_j^{(1)}, s' \in V_{j-1}^{(1)}\}$
  return $\mathcal{G}^{(1)} = (\bigcup_{j \in [1..s_1^{(1)}]} V_j^{(1)}, E)$

3.4 From step i to step $i+1$

After step i we know the graph $\mathcal{G}^{(i)}$ encoding every P_i-stack configuration of σ and we want to compute the graph $\mathcal{G}^{(i+1)}$ encoding P_{i+1}-stack configurations of σ at step $i+1$. From Lemma 22 we have to check the accessibility of pushall stack configuration of $\sigma^{(i+1)}$ from P_i-stack configurations of σ. We want to avoid to check every pair of configurations (c, c') with c being a P_i-stack configuration and c' be a pushall stack configuration of $\sigma^{(i+1)}$ because the number of such pair of configurations is possibly exponential. Thus our algorithm focuses not on stack configurations of some $\sigma^{(\ell)}$ but on sets of stack configurations of blocks $B_j^{(\ell)}$, making use of Lemma 15. Using Lemma 22, Lemma 15 can be rephrased as:

Lemma 23. *Let c' be a total stack configuration of $\sigma^{(i+1)}$, $p = p^{(i)}$ and $q = q^{(i+1)}$. Then c' is a P_{i+1}-stack configuration of σ if and only if:*

- *For any $j \leq q$, $c'_{|B_j^{(i+1)}}$ is a pushall stack configuration of $\sigma_{|B_j^{(i+1)}}$, and*
- *There exists a P_i-stack configuration c of σ such that :*
  - *$c'_{|B^{(i)}_{\min(p,q)} \cup \cdots \cup B^{(i)}_q}$ is accessible from $c_{|B^{(i+1)}_{\min(p,q)} \cup \cdots \cup B^{(i+1)}_p}$ for $\sigma_{|B^{(i)}_p \bigcup B^{(i+1)}_q}$ and*
  - *$c'_{|B^{(i+1)}_1 \cup \cdots \cup B^{(i+1)}_{\min(p,q)-1}} = c_{|B^{(i)}_1 \cup \cdots \cup B^{(i)}_{\min(p,q)-1}}$*

Recall that a P_i-stack configuration of σ is encoded by a path in the sorting graph $\mathcal{G}^{(i)}$, corresponding to the $\ominus$-decomposition of the permutation $\sigma^{(i)}$ into blocks $B_j^{(i)}$. The last point of Lemma 23 ensures that the first levels (1 to $min(p^{(i)}, q^{(i+1)}) - 1$) are the same in $\mathcal{G}^{(i+1)}$ than in $\mathcal{G}^{(i)}$. The first point of Lemma 23 ensures that the last levels ($> q^{(i+1)}$) of $\mathcal{G}^{(i+1)}$ form a complete graph whose vertices are all pushall stack configurations of corresponding blocks. So the only unknown levels for $\mathcal{G}^{(i+1)}$ are those between $\min(p^{(i)}, q^{(i+1)})$ and $q^{(i+1)}$ and we can compute them by testing accessibility.

There are differents cases depending on the relative values of $p^{(i)}$ and $q^{(i+1)}$. To lighten the notations in the following, we sometimes write p (resp. q) instead of $p^{(i)}$ (resp. $q^{(i+1)}$).

3.4.1 Case $p^{(i)} = q^{(i+1)}$

If $p^{(i)} = q^{(i+1)}$ then $B^{(i+1)}_{q^{(i+1)}} \cap A^{(i)} = B^{(i)}_{p^{(i)}} \cap A^{(i)}$ (see Figure 5).

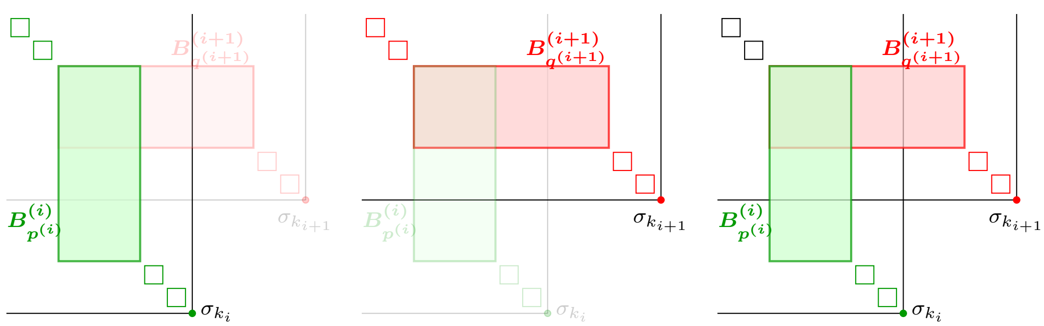


Figure 5: Block decomposition of $\sigma^{(i)}$ and of $\sigma^{(i+1)}$ when $p^{(i)} = q^{(i+1)}$

We have the sorting graph $\mathcal{G}^{(i)}$ encoding all P_i-stack configurations of σ and we want to compute the sorting graph $\mathcal{G}^{(i+1)}$ encoding all P_{i+1}-stack configurations of σ assuming that $p^{(i)} = q^{(i+1)} = min(p^{(i)}, q^{(i+1)})$.

In this case, from Lemma 23 we only have to check accessiblity of pushall configurations of $B_q^{(i+1)}$ from configurations of $B_p^{(i)}$ belonging to level p of $\mathcal{G}^{(i)}$. Indeed from the definition of a sorting graph given p.14, for any vertex v of $\mathcal{G}^{(i)}$ there is a path between vertices of $V_1^{(i)}$ and $V_{s_i}^{(i)}$ going through v, and such a path corresponds to a P_i-stack configuations of σ. Thus for any configurations x of $B_p^{(i)}$ belonging to a vertex v of level p of $\mathcal{G}^{(i)}$, there is at least one P_i-stack configurations c of σ such that $c_{|B_p^{(i)}} = x$, and $c_{|B_1^{(i)} \cup \cdots \cup B_{\min(p,q)-1}^{(i)}}$ is encoded by a path from v to level p of $\mathcal{G}^{(i)}$ (which go through each level $< p$).

If there is no pushall configuration of $B_q^{(i+1)}$ accessible from some configurations of $B_p^{(i)}$ belonging to level p of $\mathcal{G}^{(i)}$, or if $\sigma^{(i+1)}$ has no pushall configuration, then σ has no P_{i+1}-stack configuration and $\sigma_{\leq k_{i+1}}$ is not sortable (from Lemma 21).

This leads to the following algorithm:

Algorithm 5: *iteratepEqualsq*$(\mathcal{G}^{(i)})$

Data: σ a permutation and $\mathcal{G}^{(i)}$ the sorting graph at step i
Result: `false` if $\sigma_{\leq k_{i+1}}$ is not sortable, the sorting graph $\mathcal{G}^{(i+1)}$ otherwise.
begin
  $\mathcal{G}$ an empty sorting graph with s_{i+1} levels;
  $\mathcal{G}' \leftarrow ComputeG1(\sigma^{(i+1)})$ (pushall sorting graph of $\sigma^{(i+1)}$) or **return** `false`;
  Copy levels $q+1 \dots s_{i+1}$ of $\mathcal{G}'$ into the same levels of $\mathcal{G}$;
  for (c, ℓ) *in level p of* $\mathcal{G}^{(i)}$ **do**
    $\mathcal{H}$ the subgraph of $\mathcal{G}^{(i)}$ induced by (c, ℓ) in levels $< p$;
    for (c', ℓ') *in level q of* $\mathcal{G}'$ **do**
      if $isAccessible(c, c', \sigma_{|B_p^{(i)} \bigcup B_q^{(i+1)}})$ **then**
        Add (c', ℓ') in level q of $\mathcal{G}$ (if not already done);
        Merge $\mathcal{H}$ in levels $\leq q$ of $\mathcal{G}$ with (c', ℓ') as origin;
  if *level q of $\mathcal{G}$ is empty* **then**
    return `false`;
  for (c', ℓ') *in level q of* $\mathcal{G}$ **do**
    Add all edges from (c', ℓ') to each vertex of level $q+1$ of $\mathcal{G}$;
  return $\mathcal{G}$

3.4.2 Case $p^{(i)} < q^{(i+1)}$

If $p^{(i)} < q^{(i+1)}$ then $B_{q^{(i+1)}}^{(i+1)} \cap A^{(i)} \subsetneq B_{p^{(i)}}^{(i)} \cap A^{(i)}$ (see Figure 6).

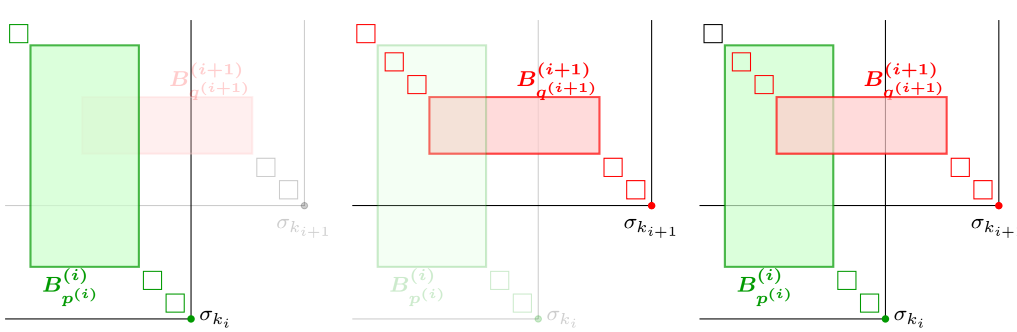


Figure 6: Block decomposition of $\sigma^{(i)}$ and of $\sigma^{(i+1)}$ when $p^{(i)} < q^{(i+1)}$

Again, Lemma 23 ensures that the first $p-1$ levels of $\mathcal{G}^{(i+1)}$ come from those of $\mathcal{G}^{(i)}$ and the levels $>q$ are all pushall stack configurations of the blocks $B^{(i+1)}_{>q}$ of $\sigma^{(i+1)}$. The difficult part is from level p to level q. As in the preceding case, by Lemma 23, we have to select among pushall stack configurations of blocks $p, p+1, \ldots, q$ of $\sigma^{(i+1)}$ those accessible from a configuration of $B^{(i)}_p$ that appears at level p in $\mathcal{G}^{(i)}$. We can restrict the accessibility test from configurations of $B^{(i)}_p$ appearing in graph $\mathcal{G}^{(i)}$ to pushall stack configurations of $B^{(i+1)}_q$. Indeed, Lemma 14 ensures that elements of blocks $B^{(i+1)}_j$ for j from p to $q-1$ are in the same stack at time t_i and at time t_{i+1}. Thus configurations of $B^{(i+1)}_j$ for j from p to $q-1$ are restrictions of configurations of $B^{(i)}_p$. We keep the same label in the vertex to encode that those configurations of $B^{(i+1)}_p, B^{(i+1)}_{p+1}, \ldots, B^{(i+1)}_{q-1}$ come from the same configuration of $B^{(i)}_p$ and we build edges between vertices of $B^{(i+1)}_{j+1}$ and $B^{(i+1)}_j$ that come from the same configuration of $B^{(i)}_p$. It is because of this case $p=q$ that we have to label configurations in our sorting graph. Indeed two different stack configurations c_1 and c_2 of $B^{(i)}_p$ may have the same restriction to some block $B^{(i+1)}_j$ but not be compatible with the same configurations, thus we want the corresponding vertices of level j of $\mathcal{G}^{(i+1)}$ to be distinct, that's why we use labels.

More precisely we have the following algorithm.

Algorithm 6: *iteratepLessThanq*$(\mathcal{G}^{(i)})$

Data: σ a permutation and $\mathcal{G}^{(i)}$ the sorting graph at step i
Result: `false` if $\sigma_{\leq k_{i+1}}$ is not sortable, the sorting graph $\mathcal{G}^{(i+1)}$ otherwise.
begin
  $\mathcal{G}$ an empty sorting graph with s_{i+1} levels;
  $\mathcal{G}' \leftarrow ComputeG1(\sigma^{(i+1)})$ (pushall sorting graph of $\sigma^{(i+1)}$) or **return** `false`;
  Copy levels $q+1, \ldots, s_{i+1}$ of $\mathcal{G}'$ into the same levels of $\mathcal{G}$;
  for (c, ℓ) *in level* p *of* $\mathcal{G}^{(i)}$ **do**
    $\mathcal{H}$ the subgraph of $\mathcal{G}^{(i)}$ induced by (c, ℓ) in levels $<p$;
    for (c', ℓ') *in level* q *of* $\mathcal{G}'$ **do**
      if $isAccessible(c, c', \sigma_{|B^{(i)}_p \bigcup B^{(i+1)}_q})$ **then**
        Add (c', ℓ') in level q of $\mathcal{G}$ (if not already done);
        for j *from* $q-1$ *downto* p **do**
          Add $(c_{|B^{(i+1)}_j}, \ell)$ in level j of $\mathcal{G}$;
          Add an edge between $(c_{|B^{(i+1)}_j}, \ell)$ and $(c_{|B^{(i+1)}_{j+1}}, \ell)$ in $\mathcal{G}$.
        Merge $\mathcal{H}$ in levels $\leq p$ of $\mathcal{G}$ with $(c_{|B^{(i+1)}_p}, \ell)$ as origin;
  if *level* q *of* $\mathcal{G}$ *is empty* **then**
    return `false`;
  for (c', ℓ') *in level* q *of* $\mathcal{G}$ **do**
    Add all edges from (c', ℓ') to each vertex of level $q+1$ of $\mathcal{G}$;
  return $\mathcal{G}$;

Note that in Algorithm 6, before calling $isAccessible(c, c', \sigma_{|B^{(i)}_p \bigcup B^{(i+1)}_q})$ we extend configuration c' to $D^{(i)} \bigcup B^{(i+1)}_q$ by assigning the same stack than in c to points of $D^{(i)} \setminus B^{(i+1)}_q$.

This is justified by Lemma 14.

3.4.3 Case $p^{(i)} > q^{(i+1)}$

If $p^{(i)} > q^{(i+1)}$ then $B^{(i)}_{p^{(i)}} \cap A^{(i)} \subsetneq B^{(i+1)}_{q^{(i+1)}} \cap A^{(i)}$ (see Figure 7).

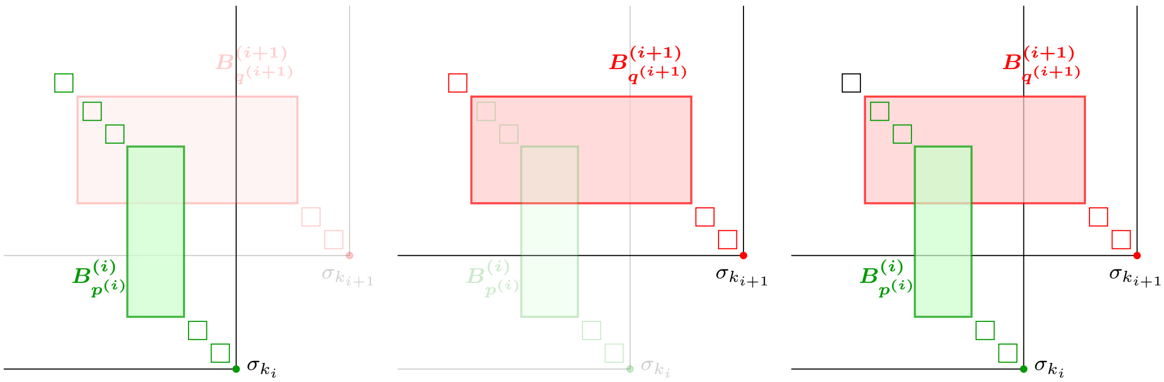


Figure 7: Block decomposition of $\sigma^{(i)}$ and of $\sigma^{(i+1)}$ when $p^{(i)} > q^{(i+1)}$

Algorithm 7: *iteratepGreaterThanq*($\mathcal{G}^{(i)}$)

Data: σ a permutation and $\mathcal{G}^{(i)}$ the sorting graph at step i
Result: `false` if $\sigma_{\leq k_{i+1}}$ is not sortable, the sorting graph $\mathcal{G}^{(i+1)}$ otherwise
begin
    $\mathcal{G}$ an empty sorting graph with s_{i+1} levels;
    $\mathcal{G}' \leftarrow ComputeG1(\sigma^{(i+1)})$ (pushall sorting graph of $\sigma^{(i+1)}$) or **return** `false`;
    Copy levels $q+1, \dots, s_{i+1}$ of $\mathcal{G}'$ into the same levels of $\mathcal{G}$;
    for (c, ℓ) *in level* p *of* $\mathcal{G}^{(i)}$ **do**
        for (c', ℓ') *in level* q *of* $\mathcal{G}'$ **do**
            if $isAccessible(c, c', \sigma_{|B_p^{(i)} \bigcup B_q^{(i+1)}})$ **then**
                if *there is a path* $(c, \ell) \leftrightarrow (c'_{|B^{(i)}_{p-1}}, \ell_1) \leftrightarrow \dots \leftrightarrow (c'_{|B^{(i)}_q}, \ell_k)$ *in* $\mathcal{G}^{(i)}$ **then**
                    Add (c', ℓ') in level q of $\mathcal{G}$ (if not already done);
                    $\mathcal{H}$ the subgraph of $\mathcal{G}^{(i)}$ induced by $(c'_{|B^{(i)}_q}, \ell_k)$ in levels $< q$;
                    Merge $\mathcal{H}$ in levels $\leq q$ of $\mathcal{G}$ with (c', ℓ') as origin;
    if *level* q *of* $\mathcal{G}$ *is empty* **then**
        return `false`;
    for (c', ℓ') *in level* q *of* $\mathcal{G}$ **do**
        Add all edges from (c', ℓ') to each vertex of level $q+1$ of $\mathcal{G}$;
    return $\mathcal{G}$;

This case is very similar to the preceding one except that $B_p^{(i)}$ is not cut into pieces but glued together with preceding blocks. As a consequence, when testing accessibility of a configuration of $B_q^{(i+1)}$, we should consider every corresponding configuration in $\mathcal{G}^{(i)}$, that is every configuration obtained by stacking configurations at level $q, q+1, \dots, p$ in $\mathcal{G}^{(i)}$. Unfortunately this may give an exponential number of configurations, but noticing that by Lemma 14 elements of blocks $B_q^{(i)}, B_{q+1}^{(i)} \dots B_{p-1}^{(i)}$ are exactly in the same stack at time t_i and at time t_{i+1}, it is sufficient to check the accessibility of a pushall configuration c' of $B_q^{(i+1)}$ from a configuration c of $B_p^{(i)}$ and verify afterwards whether the configuration c has ancestors in $\mathcal{G}^{(i)}$ that match exactly the configuration c'. This leads to the Algorithm 7.

Note that in Algorithm 7, before calling $\mathit{isAccessible}(c, c', \sigma_{|B_p^{(i)} \bigcup B_q^{(i+1)}})$ we extend configuration c to $D^{(i)} \bigcup B_p^{(i)}$ by assigning the same stack than in c' to points of $D^{(i)} \setminus B_p^{(i)}$. This is justified by Lemma 14.

Now that we have described all steps of our algorithm, we turn to the study of its complexity.

4 Complexity Analysis

In this section we study the complexity of our main algorithm: *isSortable(σ)* (Algorithm 3). The key idea for the complexity study relies on a bound of the size of each graph $\mathcal{G}^{(i)}$, as described in the following lemma.

Lemma 24. *For any $i \in [1..r]$, the maximal number of vertices in a level of $\mathcal{G}^{(i)}$ is $9n + 2$ where n is the size of the input permutation.*

Proof. From Theorem 4.4 of [3], the maximal number of pushall stack configurations of a $\ominus$-indecomposable permutation π is $9|\pi| + 2$.

By definition of $\mathcal{G}^{(1)}$, the vertices of a level correspond to pushall stack configurations of a given block of the $\oplus_1$-decomposition of the input permutation σ. Thus the cardinality of a level is bounded by $9k + 2$ where k is the size of the corresponding block. As $k \leq n$, the result holds for $i = 1$.

Suppose now that the result is true for a given $\mathcal{G}^{(i)}$, we show that it is then true for $\mathcal{G}^{(i+1)}$. The graph $\mathcal{G}^{(i+1)}$ is build from $\mathcal{G}^{(i)}$ using Algorithm 5, 6 or 7. In each case for a level j of $\mathcal{G}^{(i+1)}$ we have:

If $j > q^{(i+1)}$ then vertices of the level j of $\mathcal{G}^{(i+1)}$ are the pushall stack configurations corresponding to the block $B_j^{(i+1)}$ of the $\oplus_{i+1}$-decomposition of σ. Thus Theorem 4.4 of [3] ensures that the cardinality of level j is bounded by $9n + 2$.

If $j = q^{(i+1)}$ then vertices of the level j of $\mathcal{G}^{(i+1)}$ are a subset of the pushall stack configurations corresponding to the block $B_j^{(i+1)}$ of the $\oplus_{i+1}$-decomposition of σ. Again Theorem 4.4 of [3] ensures that the cardinality of level j is bounded by $9n + 2$.

If $j < p^{(i)}$ then vertices of the level j of $\mathcal{G}^{(i+1)}$ are a subset of vertices of the level j of $\mathcal{G}^{(i)}$. By induction hypothesis the cardinality of level j is bounded by $9n + 2$.

If $p^{(i)} \leq j < q^{(i+1)}$ then vertices of the level j of $\mathcal{G}^{(i+1)}$ are restrictions of a subset of vertices of the level j of $\mathcal{G}^{(i)}$. By induction hypothesis the cardinality of level j is bounded by $9n + 2$, concluding the proof. ☐

Lemma 25. *For any $i \in [1..r]$, the number of vertices of $\mathcal{G}^{(i)}$ is $\mathcal{O}(n^2)$ and the number of edges of $\mathcal{G}^{(i)}$ is $\mathcal{O}(n^3)$, where n is the size of the input permutation.*

Proof. The result follows from Lemma 24 as there are at most n levels and there are edges only between consecutives levels. ☐

Theorem 26. *Given a permutation σ, Algorithm 3* isSortable(σ) *decides whether σ is sortable with two stacks in series in polynomial time w.r.t. $|\sigma|$.*

Proof. Algorithm 3 involves four other subroutines: *ComputeG1* (Algorithm 4), *iteratepEqualsq* (Algorithm 5), *iteratepLessThanq* (Algorithm 6) and *iteratepGreaterThanq* (Algorithm 7).

Each for-loop in these algorithms is executed at most a linear number of time by Lemma 24.

Moreover each included operation is polynomial by Lemmas 25 and 11. □

Notice that a more precise analysis of complexity leads to an overall complexity of $\mathcal{O}(n^5)$.

References

[1] D. E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching.* Addison-Wesley, 1973.

[2] M. M. Murphy. *Restricted permutations, anti chains, atomic classes and stack sorting.* Phd thesis, University of St Andrews, 2002.

[3] A. Pierrot and D. Rossin. 2-stack pushall sortable permutations, 2013, arXiv:1303.4376.

[4] J. West. Sorting twice through a stack. *Theor. Comput. Sci.*, 117(1&2):303–313, 1993.